1 Parser

Once the actual code has been separated from proof code, the Parser then parses into an abstract syntax tree.

```
module Parser where
  import Data.Map (Map)
  import qualified Data.Map.Strict as Map
  import qualified Lexer
```

The NativeAST is, for now, a placeholder for whatever type is produced by the lexer/parser of the language being proven.

```
  data NativeAST = NativeASTNode

  parseCode :: String → AST
  parseCode code = transformAST NativeASTNode

  transformAST :: NativeAST → AST
  transformAST native = ID "The code" VEmpty
```

Once the code has been turned into a NativeAST, it is then transformed into the AST by the pluggable Transformer. Meanwhile, the proof code must also be converted into the definitions used to prove the program. These are represented by the same AST as the code, but this transformation is handled here.

```
  data AST = ID String AST -- name ArgumentList
           | ArgumentList AST AST -- Annotation ArgumentList
           | TypeOf AST -- ID
           | Annotation AST AST -- ID Type
           | Let AST AST AST -- ID Type Body
           | Function [AST] AST -- Parameters Body
           | Application AST [AST] -- Function Parameters
           | Exists AST AST AST -- ID Type Body
           | IntroExists AST AST -- Type Value [will this need another
             argument?]
           | ElimExists AST AST -- Exists Body [how does this work again?
              does it need another argument too?]
           | And AST AST -- Type Type
           | IntroAnd AST AST -- Left Right
```

```
  | ElimAndLeft AST AST -- And Body
  | ElimAndRight AST AST -- And Body
  | Or AST AST -- Type Type
  | IntroOrLeft AST AST -- Or Value
  | IntroOrRight AST AST -- Or Value
  | ElimOr AST AST AST -- Or LeftBody RightBody
  | Contradiction
  | ElimContradiction AST AST -- Contradiction Body [does this
      have a body? contradiction usually means done]
  -- value nodes
  | VInteger Int -- Value
  | VFloat Float -- Value [is this needed? or just define as a
      pair or in STL]
  | VChar Char -- Value [is this needed? or just define as an
      int or in STL]
  | VBoolean Bool -- True/False
  | VCons AST AST -- Head Tail
  | VEmpty -- empty list
  | VSymbol String -- For
  | VNull -- the empty value
  | VUndefined -- the non-existent value
  -- induction [do these need that 4th param like last time?]
  | IndInteger AST AST AST -- Int BodyS BodyZ [what if it isn't
      natural, is it the same?]
  -- [how to use a float? is float usage STL?]
  -- [how to use a char? is char usage STL?]
  | IndBoolean AST AST AST -- Bool BodyT BodyF
  | IndList AST AST AST -- List BodyL BodyE [is this correct?]
  -- [how to use a symbol?]
  -- [how to use null?]
  -- [how to use undefined?]
```

The first step in parsing the proof code is, of course, lexifying it. This step is taken on by the Lexer. After that, we move on to parsing, using the LR(1) algorithm. I think. That's what I'm going for anyway.

```
parseProofs :: String → AST
parseProofs proofText = parse $ Lexer.lexify proofText

parse :: [Lexer.Token] → AST
parse tokens = fst $ stateStart $ map Left tokens

type StateFn = [Either Lexer.Token AST] → (AST, [Either Lexer.Token
    AST])

stateStart :: StateFn
```

```
stateStart (Left Lexer.BOF : rest) = stateDefns rest

stateDefns :: StateFn
stateDefns (Left Lexer.Type : rest) = stateTypeDef rest
stateDefns (Left Lexer.Let : rest) = stateLet rest

stateTypeDef :: StateFn
stateTypeDef rest =
  let (name, rest) = stateID rest in
    let (args, rest) = stateType rest in
      let (body, rest) = stateDefns rest in
        (Let name args body, rest)

stateLet :: StateFn
stateLet rest =
  let (name, rest) = stateID rest in
    let (typ, rest) = stateStart rest in
      let (body, rest) = stateDefns rest in
        (Let name typ body, rest)

stateID :: StateFn
stateID (Left (Lexer.ID name) : rest) =
  let (args, rest) = stateTypeArgs rest in
    (ID name args, rest)

stateTypeArgs :: StateFn
stateTypeArgs (Left Lexer.LBrack : rest) = stateTypeArgs rest
stateTypeArgs (Left Lexer.RBrack : rest) = (VEmpty, rest)
stateTypeArgs rest =
  let (ann, rest) = stateAnnotation rest in
    let (end, rest) = stateTypeArgs rest in
      (ArgumentList ann end, rest)

stateAnnotation :: StateFn
stateAnnotation rest =
  let (name, rest) = stateID rest in
    let (typ, rest) = stateStart rest in
      (Annotation name typ, rest)

-- TODO: this one very complex
stateType :: StateFn
stateType _ = (ID "The type!" VEmpty, [])
```

Once parsing is complete the two trees are merged into one containing the actual code annotated by proof terms. This is the final tree which is returned to the Compiler to be used by the Analyzer in assuring that the program is valid.

```
annotates :: AST → AST → AST
annotates proof code = Annotation code proof
```